TÌM KIẾM CHUỖI

Challenge 01

Phan Lộc Sơn
19120033

Đoàn Kim Huy
19120239

Đỗ Hoài Nam
19120296

Trần Anh Huy
19120082

Cấu trúc Dữ liệu và Giải thuật
19CNTN

Khoa Công nghệ Thông tin
Đại học Khoa học Tự nhiên
Việt Nam
23 - 11 - 2020

TÌM KIẾM CHUỖI

1 Xác định vấn đề

Tìm kiếm chuỗi hiện diện rất nhiều trong cuộc sống, ví dụ như: tìm tên thầy dạy DSA trong danh sách giảng viên, tìm tên trong bảng điểm,.. hoặc trong khoa học, như là tìm liệu cấu trúc ADN của virus này có trong virus khác hay không.

Trong Tin học, các trình soạn thảo văn bản thường phải tìm kiếm (tất cả) các lần xuất hiện của một đoạn văn bản trong một văn bản dài. Thông thường văn bản được chỉnh sửa liên tục, và các phần văn bản cần tìm kiếm thì được nhập bởi người dùng. Việc phát minh ra các thuật toán tìm kiếm chuỗi hiệu quả đã giúp ích rất nhiều cho các bài toán kể trên.

Bài toán tìm kiếm chuỗi thường được mô tả theo mô hình sau:
Một chuỗi cần tìm kiếm S có dạng một chuỗi kí tự S[1..m], cần tìm chuỗi S trong một đoạn văn bản T[1..n], với m <= n. Đồng thời, tất cả các ký tự trong T và S đều thuộc về một tập hữu hạn các ký tự cho trước. Chuỗi S được gọi là xuất hiện bắt đầu từ vị trí p + 1 nếu thoả điều kiện: T[p + j] = S[j], với 1 <= j <= m.

Có 3 thuật toán thường được dùng để tìm kiếm chuỗi:

- Thuật toán Brute-force (vét cạn, hay còn gọi là thuật trâu)
- Rabin - Karp
- Knuth - Morris - Pratt

Chi tiết của từng thuật toán sẽ được trình bày bên dưới.

2 Một số thuật toán tìm kiếm chuỗi

1. Brute-force

   Giải thuật Brute-Force, hay còn gọi là vét cạn, là thuật toán đơn giản nhất trong các thuật toán tìm kiếm chuỗi con *pattern* trong chuỗi cha *text*.

   Có thể giải thích đơn giản, giải thuật Brute-Force so sánh lần lượt mỗi chuỗi con *subtext* của *text* có cùng chiều dài với *pattern* với *pattern*, nếu tìm được, trả về kết quả là vị trí được tìm thấy; khi không tìm được kết quả mong muốn, trả về giá trị quy ước là không tìm thấy.

   Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
   text = `Let them go!`
   pattern = `them`

```
Let them go!
them
```

```
Let them go!
 them
```

```
Let them go!
  them
```

```
Let them go!
   them
```

```
Let them go!
    them
```

```
Let them go!
     them
```

```
Let them go!
      them
```

```
Let them go!
       them
```

Let them go!
them

Ta tìm thấy chuỗi pattern tại vị trí thứ 4!

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Brute-Force:

```
function: Brute-Force String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
subtext: 1 đoạn thân của chuỗi cha có cùng chiều dài với pattern.
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  vị trí tìm thấy = -1
  subtext = chuỗi con đầu text có độ dài bằng pattern
  while (chưa tìm thấy hoặc chưa tới cuối text)
    if (từng ký tự của subtext = pattern):
      trả về vị trí
    else:
      dịch chuyển chuỗi con subtext trong text sang phải 1 chữ cái
  Trả về: vị trí tìm thấy
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp N - M + 2 chuỗi *subtext* đầu không trùng với *pattern*)
  → Cần M(N - M + 1) lần. Vì duyệt tới cuối mảng nên đây là trường hợp tìm thấy ở cuối mảng, hoặc không tìm thấy
  → Cận trên $O(MN)$ (vì N > N - M + 1).
  → Cấp phát bộ nhớ: 0.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự.
  → Cần M lần.
  → Cận trên $O(M)$.
  → Cấp phát bộ nhớ: 0.

Độ phức tạp của thuật toán trong trường hợp trung bình: $O(MN)$

Đánh giá:

- Dễ hiểu, thuật toán này chỉ duyệt từ đầu đến cuối, so sánh tuần tự từng chuỗi con với chuỗi cần tìm kiếm.
- Không cần bước tiền xử lý (như các thuật toán được trình bày bên dưới).
- Độ phức tạp $O(MN)$. Không cần xin thêm bộ nhớ.

2. Rabin-Karp

Thuật toán Rabin-Karp là một thuật toán được sử dụng để tìm kiếm chuỗi con *pattern* trong chuỗi cha *text* bằng cách sử dụng một hàm băm.

Hàm băm là một hàm chuyển đổi mọi chuỗi thành giá trị số, giá trị này được gọi là mã băm của nó. Ví dụ, chúng ta có thể có hàm băm hash("hello")=5.

Giống như Thuật toán Brute-Force, thuật toán Rabin-Karp cũng dịch *pattern* qua từng phần tử trong *text* để so sánh. Nhưng sự khác biệt là thuật toán Rabin-Karp so khớp mã băm của *pattern* với mã băm

của chuỗi con *subtext* của *text*, và nếu mã băm khớp thì thuật toán sẽ so sánh từng ký tự trong 2 chuỗi với nhau.

Nếu mã băm được biểu diễn bằng số nguyên không quá 64 bits, độ phức tạp thời gian (time-complexity) của việc so sánh *pattern* có độ dài m với *subtext* có cùng độ dài giảm từ O(m) xuống O(1).

Tuy nhiên mọi thứ đều có hai mặt, vấn đề của hàm băm đó là mã băm của hai chuỗi khác nhau có thể bằng nhau. Ví dụ xét hàm băm hash(S) tính mã băm của xâu S bằng cách cộng mã ASCII của các kí tự trong S: hash("abcd")=97+98+99+100=394, hàm băm này quá đơn giản và có khả năng gây trùng mã băm cao: hash("dcba")=100+99+98+97=394, nhưng "abcd" $\neq$ "dcba".

Một hàm băm tốt thoả mãn các điều kiện sau:

- Tính toán nhanh.
- Xác suất trùng mã băm nhỏ.

Thuật toán Rabin-Karp xây dựng hàm băm với ý tưởng cơ số: xem mọi xâu như là một chuỗi số với một cơ số (base) nào đó. Hàm băm được tính tương tự như việc ta chuyển một số nguyên về giá trị của nó, nếu là xâu kí tự thì có thể sử dụng mã ASCII (hoặc UNICODE). Một số ví dụ:

- base=10, hash("425")=4×102+2×101+5×100=425.
- base=26, kí tự là chữ cái từ a đến z: hash("abc")=97×262+98×261+99×260=68219.

Để tránh tràn số thì kết quả trên được chia lấy dư cho một số q, thường được chọn là một số nguyên tố lớn. Nếu gọi tập các kí tự được sử dụng trong chuỗi là $\sum$ thì base thường được chọn sao cho base=$|\sum|$ hoặc là một số nguyên tố lớn.

Độ phức tạp thời gian để tính mã băm của chuỗi độ dài k mất O(k). Khi hiện thực thuật toán, ta sẽ "trượt" *pattern* có độ dài m trên *text* từ vị trí 0 đến n-m để so sánh mã băm. Rolling hash là hàm băm có thể tính mã băm h_i của *text*[i... i+m-1] dựa trên mã băm h_{i-1} của *text*[(i-1)... (i+m)] chỉ trong thời gian O(1) thay vì tính lại trong thời gian O(m), từ đó tăng tính hiệu quả.

$h_i = (base \times (h_{i-1} - base^{m-1} \times text[i-1]) + text[i+m-1])\%q$

Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
Để đơn giản, ta chọn tập các kí tự được sử dụng trong chuỗi là {a, b, c, d, e, f, g, h, i, j} ứng với giá trị số lần lượt là {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} và base=10, q=13.
text = abccddaefg
pattern = cdd
Mã băm của *pattern*: hash("cdd")=$(3 \times 10^2 + 4 \times 10^1 + 4 \times 10^0)\%13 = 344\%13 = 6$

---

abccddaefg // hash("abc")=$(1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0)\%13 = 123\%13 = 6$
cdd // Vì mã băm của *subtext* "abc" và *pattern* bằng nhau
// nên thuật toán so sánh từng ký tự của chúng.
// Do "abc"$\neq$*pattern* nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("bcc")=$(10 \times (123 - 10^2 \times 1) + 3)\%13 = 233\%13 = 12$
cdd // Vì mã băm của *subtext* "bcc" và *pattern* khác nhau
// nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("ccd")=$(10 \times (233 - 10^2 \times 2) + 4)\%13 = 334\%13 = 9$
cdd // Vì mã băm của *subtext* "ccd" và *pattern* khác nhau
// nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("cdd")=$(10 \times (334 - 10^2 \times 3) + 4)\%13 = 344\%13 = 6$
cdd // Vì mã băm của *subtext* "ccd" và *pattern* bằng nhau
// nên thuật toán so sánh từng ký tự của chúng.
// Do "abc"=*pattern* nên chuỗi đã được tìm thấy.

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Rabin-Karp:

```
function: Rabin-Karp String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
subtext: 1 đoạn thân của chuỗi cha có cùng chiều dài với pattern.
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  n=chiều dài chuỗi text
  m=chiều dài chuỗi pattern
  Tính mã băm của pattern và chuỗi con đầu text có độ dài bằng pattern
  Dịch pattern từ vị trí 0, qua từng phần tử của text đến vị trí n-m
    Kiểm tra mã băm của subtext hiện tại và pattern
      Nếu bằng nhau, so sánh từng ký tự trong 2 chuỗi
        Nếu hoàn toàn giống nhau, trả về vị trí
    Tính mã băm của subtext tiếp theo, nếu âm cộng thêm một lượng q
  Nếu trong vòng lặp không tìm thấy pattern, ta đã duyệt hết text, trả
     về -1 báo kết quả không tìm thấy pattern trong text.
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N:
Phân tích độ phức tạp của thuật toán:

Trong quá trình tiền xử lý, để tính mã băm cho *pattern* cần duyệt qua M phần tử, tương tự với *subtext* đầu tiên *text*, vậy độ phức tạp là $O(M + M) \in O(M)$.

Trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*, khi đó chỉ cần tốn M lần so sánh các ký tự, kết hợp với quá trình tiền xử lý nên độ phức tạp là $O(M + M) \in O(M)$.

Trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp mã băm bằng nhau và cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp chuỗi cần tìm nằm cuối, N - M + 2 chuỗi *subext* đầu không trùng với *pattern*).
  $\rightarrow$ Cần nhiều nhất M(N - M + 1) lần so sánh. Vì duyệt tới cuối chuỗi nên đây là trường hợp tìm thấy ở cuối hoặc không tìm thấy và xảy ra trùng mã băm ở tất cả lần so sánh trước đó.
  $\rightarrow$ Cận trên $O(MN)$ (vì N $>$ N - M + 1).
  Trường hợp trung bình: $O(N + M) \in O(M)$

Do trong quá trình tìm kiếm có thực hiện tính toán và lưu mã băm nên chi phí bộ nhớ là hằng số $O(1)$.
Đánh giá:

- Mặc dù trong trường hợp xấu nhất độ phức tạp là $O(MN)$ không tốt hơn giải thuật Brute-Force, nhưng giải thuật Rabin-Karp hoạt động tốt hơn nhiều trong trường hợp trung bình và thực tế.
- Có bước tiền xử lý với độ phức tạp $O(M)$ trước khi bắt đầu tìm kiếm.
- Độ phức tạp $O(MN)$. Chi phí bộ nhớ $O(1)$.

3. Knuth-Morris-Pratt

Giải thuật Knuth-Morris-Pratt, về cơ bản cũng giống như thuật toán Brute-Force, tuy nhiên, chỉ khác ở chỗ, Brute-Forch khi so sánh *pattern* và *subtext*, Brute-Force so sánh toàn bộ các ký tự của chúng lại từ đầu, còn Knuth-Morris-Pratt, từ lần so sánh trước, sẽ quyết định có so sánh pattern với *subtext* kế tiếp không, và nếu không sẽ nhảy bao nhiêu bước đến *subtext* khác. (đã biết được chúng giống nhau), từ đó tiết kiệm được chi phí giải bài toán.

Để dễ hiểu, ta xét 1 ví dụ như sau:

text = ababcababd
pattern = ababd
Quy ước vị trí ký tự đầu tiên trong chuỗi là 1.

```
ababcababd
ababd            // Bắt đầu so sánh pattern với subtext đầu tiên của text tương tự giải thuật Brute-Force.
```

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd // Ta thấy ký tự thứ 5 của *pattern* và *subtext* không khớp nhau
// nên ta sẽ tìm *subtext* tiếp theo để so sánh.
// Ta nhận thấy rằng 4 ký tự đầu của *pattern* khớp với *subtext*,
// trong 4 ký tự đó (abab), có có **2 ký tự đầu** giống **2 ký tự cuối**.
// Ta không nên chọn *subtext* tiếp theo là a**babca**babd
// vì *subtext* của bước so sánh này luôn không trùng *pattern*.
// Như vậy, lần tiếp theo ta nên so sánh *pattern* với chuỗi ab**abcab**abd
// vì chuỗi dài nhất để chuỗi tiền tố *pattern*[1, 4] và chuỗi hậu tố *pattern*[2, 4]
// giống nhau là **ab** có độ dài là 2.

---

ababcababd
abab d // Ta lấy pattern[1, 4] để tìm chuỗi chung là vì nó là phần đã được so sánh đúng
// ở lượt trước và ta cần tái sử dụng các phép so sánh đúng này.
// Ngoài ra, chuỗi hậu tố ta chỉ lấy của *pattern*[2, 4] vì bước tiếp theo
// ta sẽ dịch *pattern* để tiếp tục tìm kiếm
// (nếu lấy *pattern*[1, 4] mà cả 4 ký tự giống nhau thì bước tiếp theo
// ở giải thuật này ta lại ở vị trí hiện tại)
// Ta có: pattern[1, 3] = text[2, 4], mà text[2, 4] = pattern[2, 4] (lượt so sánh trước)
// như vậy là: pattern[1, 3] = pattern[2, 4], tức là có chuỗi độ dài 3 vừa là chuỗi đầu
// vừa là chuỗi cuối của pattern[1, 4] (vô lý vì 2 là số lớn nhất).
// Ta so sánh *pattern* với *subtext* bắt đầu bằng ký tự thứ 3
// (vì 2 ký tự đầu tiên đã đúng).
// Vị trí không khớp là *pattern*[3], ta thấy không có chuỗi chung vừa là
// chuỗi đầu và vừa là chuỗi cuối của pattern[1, 2]
// nên ta dịch *pattern* sang phải 1 ký tự để tiếp tục so sánh.

---

ababcababd
abab d // Tiếp tục so sánh *pattern* với *text* tương tự giải thuật Brute-Force.

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd // *pattern* khớp *subtext* tại vị trí thứ 6 của *text*
// Vị trí tìm được là 6, kết thúc tìm kiếm.

Note: Với string là chuỗi có độ dài n. và j là số nguyên dương $0 < j < n + 1$
Chuỗi đầu: Prefix, có dạng prefix[] = string[1, j], chuỗi cuối: Suffix, có dạng Suffix[] = string[j, n].

Từ ví dụ trên, ta tìm cách thiết kế ý tưởng của KMP để tìm chuỗi 1 cách tổng quát:

- Ta thấy rằng, để KMP tìm cách nhảy hiệu quả, cần phải xử lý trước pattern, tìm chuỗi chung vừa là chuỗi đầu, vừa là chuỗi cuối của pattern. Bước này gọi là bước tìm Failure Function, tại mỗi vị

trí j trong pattern ($0 < j < len(pattern) + 1$), tìm độ dài lớn nhất của chuỗi chung và lưu vào mảng $f[]$ để lưu trữ.

- Khi đang so sánh pattern và subtext, ta có thể gặp các trường hợp ($0 < i <= len(subtext)$):
  1. Nếu pattern[i] = subtext[i], tăng thêm i để so sánh ký tự kế tiếp.
  2. Nếu pattern[i] $\neq$ subtext[i], và i = 1, tức là ký tự đầu của chúng khác nhau, đơn giản là chỉ cần nhảy subtext sang bên phải 1 bước.
  3. Nếu pattern[i] $\neq$ subtext[i], và i > 1, tức là đã có 1 lượng i - 1 ký tự ở trước đã được so sánh đúng, bây giờ là đến lúc cần dùng mảng f đã tính trước ở bên trên, ta lấy f[i - 1] (chính là độ dài lớn nhất của chuỗi chung của pattern[1, i - 1]) và ta sẽ nhảy subtext 1 lượng min((i - 1) - f[i - 1], 1) để chuẩn bị cho lượt so sánh kế tiếp.

Từ ý tưởng trên, ta xây dựng mã giả cho thuật toán KMP:

```
function: Build Failure Function
pattern: chuỗi cần tìm
Trả về: Mảng f lưu độ dài lớn nhất f[i] của chuỗi chung prefix và suffix
   của pattern[1, i].

  f[] = 0 để lưu mảng trả về
  fpre = 0;   // lưu f[i - 1] trong vòng lặp bên dưới
  i = 2;    // Ta chỉ tính f[2] trở đi, f[1] luôn bằng 0
  while (i <= n)
    Nếu pattern[i] == pattern[f[i - 1] + 1]
      fpre++
      f[i] = fpre // tận dụng kết i - 1 kết quả so sánh của f[i - 1], ta
         chỉ cần so sánh 1 ký tự.
      i++
    Nếu pattern[i] != pattern[f[i - 1] + 1] và f[i - 1] = 0
      // Tức là suffix và prefix là chuỗi có độ dài f[i - 1] + 1 có ký
         tự cuối không giống
      // -> f[i] = 0
      f[i] = 0
      i++
    Nếu pattern[i] != pattern[f[i - 1] + 1] và f[i - 1] > 0
      // Trường hợp này, có thể f[i] = f[i - 1]
      // Ta sẽ tận dụng pattern[1, f[i- 1]] = pattern[i - f[i-1], i]
      // Và f[f[j]] (độ dài này lớn nhất là j - 1)
      // Ví dụ AAABAAAA khi xét i = 8, ta lợi dụng chuỗi [AA]ABAAAA =
         A[AA]BAAAA và [AAA]BAAAA = AAAB[AAA]A
      // Suy ra được [AA]ABAAAA = A[AA]BAAAA = AAABA[AA]A
      // Như thế chỉ cần xét ký tự AA[A]BAAAA và AAABAAA[A] để xác định
         f[i]
      fpre = f[fpre]
  Trả về: f[]

function: KMP String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  i = 1; j = 1; (i đánh dấu vị trí đang duyệt trong text, j đánh dấu vị
     trí đang duyệt trong pattern)
  while (i < len(text) - len(pattern))
    Nếu pattern[j] == text[i]
      Tiếp tục i++ và j++ và so sánh
        Nếu j = len(pattern) trả về đã tìm thấy pattern tại vị trí i - j.
    Nếu pattern[j] != text[i] và j == 1
      i = i + 1  //Chuyển nhảy subtext sang phải 1 đơn vị
    Nếu pattern[j] != text[i] và j > 1
      j = f[i] và tiếp tục so sánh text[i] và pattern[j] // sau khi gán
         j = f[i] (giảm j) thì ta đã cố tình dịch chuyển subtext đang
         so sánh sang subtext khác!
```

```
41          Nếu trong vòng lặp không tìm thấy pattern, ta đã duyệt hết text, trả
                về -1 báo kết quả không tìm thấy pattern trong text.
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Trong quá trình tiền xử lý (tính hàm Failure Function), trong vòng while chỉ tốn nhiều nhất 1 - 2 lần để tính f[i], vậy độ phức tạp là $O(M)$
- Trong quá trình so sánh, mỗi lần, ta hoặc sẽ tăng i lên 1 đơn vị, hoặc sẽ giảm j ít nhất 1 đơn vị (trong trường hợp dịch j), nghĩa là tăng k lên 1 đơn vị, k = i - j < N. Vậy tổng cộng trong vòng lặp có thể lặp < 2n lần (N lần tăng i, ít hơn N lần tăng k), như vậy độ phức tạp là $O(2N)$.
  → Độ phức tạp $O(M + N)$.
  → Cấp phát bộ nhớ: $O(M)$.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự. Ngoài ra còn tốn thêm chi phí tính Failure Function $O(M)$.
  → Cận trên $O(M)$.
  → Cấp phát bộ nhớ: $O(M)$.

Độ phức tạp của thuật toán trong trường hợp trung bình:

- Cận trên $O(N + M)$.
- Cấp phát bộ nhớ: $O(M)$.

Đánh giá:

- Nhanh, chi phí tuyến tính.
- Phải xử lý trước khi tìm kiếm thực sự.
- Độ phức tạp $O(M + N)$. Chi phí bộ nhớ $O(M)$.

TÌM KIẾM CHUỖI TRONG TIN HỌC

Ví dụ về Crossword game

1 Hướng giải Crossword game

Trong phần này, nhóm chúng tôi quyết định sẽ sử dụng thuật toán KMP để làm giải thuật tìm kiếm chuỗi, bởi vì giải thuật này có vẻ nhanh hơn 2 giải thuật còn lại với thời gian $O(M + N)$.

Thuật toán chính của chương trình là đọc mảng chữ cái vào mảng **table** để tìm theo hàng ngang, và tạo mảng **Transpose_table** bằng cách chuyển vị mảng **table** để tìm theo hàng dọc.

Tìm bằng thuật toán KMP, hàng ngang trước và hàng dọc sau, nếu tìm thấy, in ra vị trí tìm thấy, nếu không, in ra NF.

Độ phức tạp:

- Có H hàng, mỗi hàng cần tìm 1 chuỗi có độ dài M trong chuỗi W, vậy độ phức tạp là $O(H * (M + W))$.

- Có W cột, mỗi cột cần tìm 1 chuỗi có độ dài M trong chuỗi H, vậy độ phức tạp là $O(W * (M + H))$.
  $\rightarrow$ Như vậy, độ phức tạp là: $O(2WH + M(H + W))$ đối với từng từ cần tìm kiếm.

2 Một số mẫu kiểm tra lời giải

Dưới đây là bảng số liệu nhóm chúng tôi đã thực hiện trên các tập dữ liệu $n \times n$: 5, 10, 20, 50:

- Bộ dữ liệu ngẫu nhiên:

Size	BF	RK	KMP
5	125	178	146
10	135	147	168
20	264	175	198
50	649	218	181

Bảng 1: Số liệu với bộ test được sinh ngẫu nhiên, đơn vị micro second.

Nhận xét: Ở các bộ dữ liệu nhỏ, Naive nhanh hơn Rabin-Karp và Knuth-Morris-Pratt vì không có bước tiền xử lý.

- Bộ dữ liệu xấu nhất:

Size	Naive	Rabin-Karp	Knuth-Morris-Pratt
5	747	93	129
10	1052	102	148
20	3246	125	139
50	8796	222	205

Bảng 2: Bảng số liệu với bộ test được sinh xấu nhất, đơn vị micro second.

Nhận xét: Tại bộ dữ liệu xấu nhất, Knuth-Morris-Pratt và Rabin-Karp nhanh hơn Naive
Ngoài ra, càng tăng size bộ dữ liệu, tốc độ tăng thời gian của Naive > Rabin-Karp > Knuth-Morris-Pratt (Naive tăng nhanh nhất, Knuth-Morris-Pratt tăng chậm nhất)